Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Sami Jaktholm

Building a Scalable Simulation Platform for Prototyping Distributed Machine Learning Solutions

Master's Thesis
Espoo, June 30, 2019

Supervisor: Professor Jukka Suomela
Advisor: M.Sc. Perttu Ranta-aho

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Sami Jaktholm

Title:
Building a Scalable Simulation Platform for Prototyping Distributed Machine Learning Solutions

Date: June 30, 2019 **Pages:** 80

Major: Computer Science **Code:** SCI3042

Supervisor: Professor Jukka Suomela

Advisor: M.Sc. Perttu Ranta-aho

Distributed machine learning techniques are gaining popularity in the industry. In distributed learning, client endpoints use local data to train machine learning models and potentially collaborate with one another. Currently, most development tools are designed to operate in a centralized environment. Therefore, most tools are unsuitable for development of distributed solutions. This work builds a simulation platform that allows developers simulate distributed machine learning systems in the cloud. We implement the platform on Amazon Web Services and define an iterative execution model for simulations. Our experiments show that the platform scales to more than 10,000 simulated client endpoints, is cost effective and easy to use. Most importantly, it enables the development of distributed machine learning solutions in a cloud environment.

Keywords: distributed systems, distributed machine learning, simulation, simulation platform, endpoint detection and response

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu DIPLOMITYÖN
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma TIIVISTELMÄ

Tekijä:	Sami Jaktholm		
Työn nimi:	Skaalautuva simulaatioalusta hajautettujen koneoppimisratkaisuiden kehittämiseen		
Päiväys:	30. kesäkuuta 2019	**Sivumäärä:**	80
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Professori Jukka Suomela		
Ohjaaja:	Diplomi-insinööri Perttu Ranta-aho		

Hajautettujen koneoppimismenetelmien suosio on kasvussa. Hajautetussa koneoppimisessa suuri määrä päätelaitteita kouluttaa koneoppimismalleja paikallisella tiedolla ja tekee mahdollisesti yhteistyötä keskenään. Suurin osa nykyisistä kehitystyökaluista on suunniteltu keskitettyihin järjestelmiin, eivätkä ne sovellu hajautettujen koneoppimisratkaisujen kehittämiseen. Tämä työ kehittää simulaatioalustan, jolla voidaan simuloida hajautettuja koneoppimisjärjestelmiä keskitetyssä pilviympäristössä. Alusta on rakennettu Amazon Web Services -pilvipalvelun päälle, ja työ esittää simulaatioille iteratiivisen suoritusmallin. Kokeemme näyttävät, että alusta skaalautuu yli 10 000 päätelaitteen simulointiin, se on kustannustehokas ja helppokäyttöinen. Ennen kaikkea alusta mahdollistaa hajautettujen koneoppimisratkaisujen kehittämisen pilviympäristössä.

Asiasanat:	hajautetut järjestelmät, hajautettu koneoppiminen, simulaatioalusta, uhkien havainnointi ja torjuminen
Kieli:	Englanti

Acknowledgements

First, I would like to thank my employer, F-Secure, for giving me the opportunity to work on this thesis full time for six months. I would have not finished the work if it wasn't for their generous time allocation. Special thanks go to my advisor Perttu and manager Klas for listening to my struggles and offering guidance throughout this work. Second, I would like to thank my colleagues Dmitriy and Janne for their help and comments on my work. I would also thank other fellows of F-Secure Artificial Intelligence Center of Excellence for pushing me to forget my pre-thesis responsibilities. Finally, I would like to thank my supervisor Prof. Jukka Suomela for his invaluable guidance and encouragement during the process.

Espoo, June 30, 2019

Sami Jaktholm

Abbreviations and Acronyms

APT	Advanced Persistent Threat
AWS	Amazon Web Services
AZ	Availability Zone
DML	Distributed Machine Learning
EDR	Endpoint Detection and Response
IDS	Intrusion Detection System
RDS	Rapid Detection Service
RDR	Rapid Detection & Response
UEBA	User and Entity Behavior Analytics

Contents

Chapter 1

Introduction

1.1 Background

The term Endpoint Detection and Response (EDR) is used to describe a new class of tools that detect suspicious activity in computer systems [19]. According to a 2018 study of 477 organizations, the average time to discover a breach is 197 days [38]. EDR solutions can help organizations detect and respond to breaches in a timely manner. These tools monitor and analyze endpoint behavior to find signs of abuse. If abuse is detected, EDR tools can also provide means to respond to the incident. EDR solutions often complement traditional anti-virus tools by increasing visibility into the system behavior.

The F-Secure Rapid Detection and Response Service (RDS) [29] is a managed EDR service. In RDS, lightweight sensors collect behavioral data from customer network and endpoint devices. A backend analyzes the data in real time with a combination of machine learning, statistical analysis and expert systems. If the backend detects an anomaly, it sends an alert to the Rapid Detection & Response Center (RDC). The RDC experts review all alerts, inform the customer about real incidents and provide instructions for a response [29].

1.2 Problem

The RDS backend analyzes tens of thousands of events each second. Operating at such a high scale can be challenging. First, the backend requires considerable amount of computational resources. The data volumes must be reduced to keep the backend costs at an acceptable level. Second, such a high scale constrains the analysis. For example, the context that can be considered during analysis is limited. Context is especially important when detecting malicious behavior. It is normal for a developer to execute commands via the command prompt but extremely suspicious if similar activity occurs on an endpoint from the marketing department. This approach is called *User and Entity Behavior Analytics* (UEBA) [71]. Creating, managing and using per-user behavioral profiles is difficult at the scale of the RDS backend due to resource requirements of per-user profiling.

These challenges could be solved by offloading computation to the endpoint sensors. Instead of processing all the events in the backend, endpoint sensors could use local logic and only send relevant events to the backend. Such logic could include 1) discarding events based on rules and models, 2) aggregating events and sending aggregated summaries to the backend, or 3) modeling endpoint behavior locally. Local logic can use sensor specific context more extensively than the backend could.

Unfortunately, these changes would also impact the machine learning components of the RDS backend. Currently, the backend trains machine learning models with the data the sensors send. If sensors send less data, the backend has less information available to train the models. This can decrease model accuracy. Distributed machine learning techniques, such as federated learning [44], could mitigate these issues. Distributed learning techniques allow client endpoints train machine learning models without collecting all the data into a central repository. Models running on the endpoints have access to all the data – not just a subset the sensors send to the backend.

1.3 Objective

The development of distributed learning solutions is not a trivial task. First, it is difficult to experiment with ideas and iterate on promising ones without proper tooling. Currently, most machine learning tools are designed to operate in a centralized environment. They are not suitable for development of distributed solutions. Second, the local logic must be extensively validated to ensure it generalizes to the entire customer base. F-Secure RDS customers range from small businesses to large enterprises. The endpoint behavior varies both within a single organization and between organizations. Hence, it must be easy to verify that the proposed solutions work for all F-Secure RDS customers.

The goal of this thesis is to build a simulation platform that allows F-Secure experts develop distributed solutions in a centralized cloud environment. The experts should be able to 1) quickly experiment with new ideas, 2) iterate on promising ones, and 3) validate them across the entire F-Secure RDS customer base. The platform should provide data from real sensors as input to simulations. It should scale to simulations with thousands of sensors. Simulated sensors should be able to communicate with a server component or directly with one another. Overall, the platform should enable development of distributed machine learning solutions in a centralized cloud environment.

1.4 Outline

This thesis is structured as follows. Chapter 3 presents the motivation, requirements and goals of the simulation platform in more detail. Chapters 4 and 5 document the platform implementation: Chapter 4 surveys available technologies and Chapter 5 describes the platform implementation. Finally, we evaluate the platform in Chapter 6 and conclude this thesis in Chapter 7. The next chapter provides background information on distributed machine

learning techniques and how machine learning is used in cybersecurity. We also discuss the F-Secure RDS solution in more detail.

Chapter 2

Preliminaries

2.1 Distributed Machine Learning

The amount of data has exploded over the years. While this has enabled many machine learning applications, it has also created new challenges. One of them was that machine learning algorithms were not able to scale to utilize all the available data [63]. This led to the development of *large scale learning*, a new area of machine learning that focused on scaling machine learning algorithms to efficiently utilize large data sets [15]. In 1998, Provost et al. [65] provided three approaches for scaling up machine learning: 1) make the algorithm fast, 2) relationalize the data, and 3) partition the data. The third approach, partitioning the data, was a perfect match with distributed computing. If the data set was partitioned, learning could be distributed over multiple processors.

2.1.1 History of Distributed Machine Learning

Methods for distributed learning have been developed since mid-1990s. In 1994, Provost and Hennesey [64] proposed to use unused workstation capacity for machine learning tasks on large data sets. They partitioned the data set into smaller subsets and distributed these subsets to available workstations.

Each workstation executed a machine learning program on a subset of the data and communicated partial results to other workstations. Finally, a coordinator service constructed a global model from results generated by participating workstations [64].

Provost and Hennesey designed their distributed learning system with an assumption that the workstations participating in the process were part of a reliable, high-speed network [64]. The advances in the development of wired and wireless networks led to proliferation of distributed computing systems with varying levels of connectivity [59]. The connectivity of these systems had different characteristics in terms of availability, reliability and bandwidth.

Push Towards the Edge

In early 2000s, the distributed data mining community drove the development of methods to utilize distributed computing environments with an unreliable network for machine learning. They observed that a centralized solution where mission critical data is uploaded to a central data warehouse for analysis was not suitable for emerging distributed applications. The process had latency, resource utilization was poor and centralized data mining algorithms were not suitable for distributed environments [59].

MobiMine [42] let users monitor stock market development from their mobile phones or personal digital assistants (PDAs). The MobiMine server monitors stock market data and collects it from different sites. The MobiMine clients fetch this information from the server and filter it to provide users the information that is relevant to their interests. The client software builds a model of the stock risk level the user is comfortable with and shows them information on stocks that match their risk level [42]. This provided each user a customized experience without a significant investment in server-side resources.

VEDAS [41], a vehicle data stream mining system, allowed a central operator to continuously monitor the performance and condition of their entire

vehicle fleet. As sensors on board vehicles generate massive amounts of data, collecting that into a central location was infeasible. Instead, VEDAS mined the stream of data on an on-board PDA and extracted important features from the stream. Extracted features would then be sent to a central monitoring system for analysis [41].

Large Scale Learning in Data Centers

Improvements in networking technologies reshaped the distributed machine learning landscape. The focus of development shifted from deploying machine learning solutions to the edge to collecting all the data into a data center for analysis. MapReduce [23] provided a framework for distributing computation, including machine learning, to a large set of commodity machines in a data center. The MapReduce paradigm underpins many modern general-purpose distributed computing frameworks such as Apache Hadoop [76] and Apache Spark [88]. Distributed machine learning capabilities have been built into these frameworks in projects such as Apache Mahout [77] for Hadoop and Spark MLLib [49] for Spark.

Most general-purpose distributed computing frameworks mandate synchronous communication between participating nodes. Synchronous communication is sufficient when models are simple, and the process includes only a handful of compute nodes. Synchronous communication becomes a bottleneck when the complexity of models increases and the number of nodes participating to the process grows. To solve this issue, researchers at Google, Baidu and Carnegie Mellon University developed a novel parameter server framework to facilitate asynchronous communication between nodes participating to the training process [46]. The parameter server framework stores parameters in a distributed key-value store as sparse vectors. In addition to providing standard get and put routines of a key-value store, the parameter server can provide additional vector operations to optimize parameter updates [46]. Experiments show that the parameter server framework is ef-

fective in distributed machine learning tasks with over 600TB of real data that contains hundreds of billions of samples and dimensions [47].

Federated Learning

Federated learning [44, 45] is a recently developed approach for training a machine learning model without sending all the training data to a centralized service. In federated learning, client nodes collaborate to build a globally optimized model. Clients use local data to compute model updates. These updates are then sent to a coordinator service which combines updates from multiple clients to improve the global model. Finally, the coordinator sends the updated model parameters back to clients. This process is repeated until the model is deemed to be accurate enough [44].

Federated learning provides two important advantages over methods that collect all training data into central storage. First, federated learning does not require huge compute and storage capacity [44]. The cost of compute and storage is distributed among clients participating in the process instead. Second, federated learning increases user privacy as training data never leaves the client device. The model updates sent to a coordinator server contain less information than the private training data. Furthermore, the updates are ephemeral, and they can be discarded as soon as they are folded into the global model [44]. Additional techniques such as secure aggregation [14] or differential privacy [1] can further improve user privacy in federated settings.

Peer-to-Peer Learning

Peer-to-peer learning offers a fully decentralized approach to distributed machine learning. Unlike federated learning, peer-to-peer learning techniques do not require a central coordinator service. Instead, agents participating to the training process communicate directly with one another to share information needed to improve the model [56, 83].

The research community has investigated peer-to-peer approaches to machine learning for decades. In mid-1990s, the reinforcement learning commu-

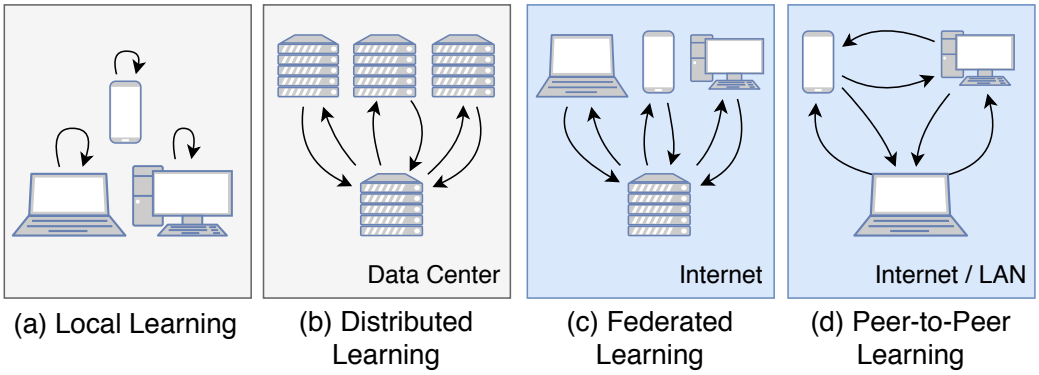


Figure 2.1: Architectures of distributed machine learning systems.

nity worked to solve the problem of optimization in a multi-agent cooperative environment [74]. The goal of these agents was to find an optimal solution to the given problem by cooperating with other agents in the environment. In 2010, Nedic et al. [56, 57] developed algorithms for finding a globally optimized model in a multi-agent network. Their work showed it was possible to find globally optimized solution to an optimization problem in a fully decentralized multi-agent environment. More recent work by Vanhaesebrouck et al. [83] focuses on building personalized models in a peer-to-peer collaboration environment.

2.1.2 Architecture of Distributed Machine Learning Systems

We discussed various approaches to distributed training of machine learning models in the previous section. We can categorize these approaches to four distinct groups:

1. In **Local Learning** [Figure 2.1(a)], each node trains a local model with the locally available data. The model training process is independent of any external services or information other nodes have.

2. In **Distributed Learning** [Figure 2.1(b)], multiple compute nodes collaborate to train a model within a data center. Each worker node

uses a subset of the data to generate model updates. A coordinator node collects the updates from the workers, folds them into a global model and distributes the updated model to the worker nodes. This process is repeated until the model is accurate enough.

3. In **Federated Learning** [Figure 2.1(c)], a large set of client devices collaborate to train a model with the help of a coordinator node. Each device uses local data to compute model updates. Like the previous case, a coordinator node collects these updates, folds them into a global model and distributes the updated model back to client devices. The biggest difference to the previous setup is that the training data never leaves the client devices.

4. In **Peer-to-Peer Learning** [Figure 2.1(d)] a large set of client devices collaborate to train a model without a central coordinator node. Clients use local data to build a model. They communicate directly with other clients to improve their local models with information other clients have acquired.

Of these four categories our work focuses on the three that include client-side computation: local, federated and peer-to-peer learning. Distributed learning in a data center is already a well-studied problem and irrelevant for the goals of this work. We use these categories to describe the architecture of distributed machine learning systems throughout the thesis.

2.2 Machine Learning and Cybersecurity

The dynamic nature and increased complexity of cyberattacks poses difficulties to traditional cyber defenses [82]. In the past two decades, machine learning has become a useful tool in detecting malware and anomalous activity in computer systems. This section provides a brief overview on how machine learning techniques are used in cybersecurity.

2.2.1 Malware Detection

Proliferation of malicious software is a major threat to security of modern computer systems. A recent study from Kaspersky reports that their products observed at least one malware threat on 22.5 % of computer systems world-wide during Q3 of 2018 [18]. Therefore, development of accurate malware detection techniques is important to computer system security. This can, however, be a daunting task. Malware authors are constantly evolving their designs to evade existing anti-malware solutions [86]. Similarly, anti-malware solution providers are constantly developing new techniques to catch more advanced malware [26].

Malware Detection Basics

Malware detection is effectively a binary classification task: given a sample, determine if the sample is malicious or not. The task can be split into two logical phases: 1) sample analysis phase, and 2) detection phase.

In the sample analysis phase, anti-malware tools extract features from the analyzed sample. Feature extraction techniques can be classified into two categories:

1. *Static analysis methods* extract information from the sample payload directly. This information includes string signatures, byte sequences and control flow graph information [31]. Static analysis methods are fast but can be evaded with binary obfuscation techniques [51].

2. *Dynamic analysis methods* collect information on the runtime behavior of a sample. That is, these techniques execute the sample and observe its behavior in an isolated sandbox environment [26]. Dynamic analysis techniques are computationally expensive but more difficult to evade.

In the detection phase, a sample is classified based on the features extracted in the sample analysis phase. Detection techniques can be classified as follows [39]:

1. *Signature-based techniques* compare the structure and behavior of a sample against a collection of patterns malware is known to exhibit. A sample is classified as malicious if it matches a malware signature. The complexity of signatures varies from simple byte sequences present in known malware to elaborate rule-based heuristics. Signature-based techniques are effective in detecting known malware but fall short in detecting previously unseen attacks [39].

2. *Anomaly-based techniques* compare the structure or behavior of a sample to a baseline generated from clean samples. A sample is classified as malicious if its structure or behavior deviates from the known baseline. Anomaly detection techniques can detect previously unseen malware, but they are prone to generate false positives [39].

Finally, hybrid systems combine multiple techniques to improve the efficiency and accuracy of malware detection [21]. These systems can use fast static analysis and signature-based detection techniques to detect known malware. If the result is inconclusive, they fall back to more expensive dynamic analysis and anomaly-based detection techniques. Consequently, hybrid systems combine the strengths of multiple analysis methods while mitigating the drawbacks individual methods have.

Machine Learning and Malware Detection

The research community has utilized machine learning techniques in malware detection since the mid-1990s [75]. Since then, machine learning techniques have been applied to a wide range of issues. These applications range from automated generation of malware signatures [36, 69] to automated classification of novel malware by analyzing their runtime behavior [17, 70].

Machine learning techniques are useful in sample analysis automation [67]. In the early days, human experts analyzed potential malware samples by hand to determine if they were malicious or not [26]. If a sample was deemed to be malicious, the experts would try to find a pattern that could

identify this sample and its variants. This task was time-consuming and error-prone [26].

In contrast to manual analysis, machine learning techniques excel in finding patterns in large datasets. The two most common machine learning concepts used in automatic malware analysis are clustering and classification [67]. Clustering enables discovery of novel malware families [13]. In clustering, potential malware samples are grouped by their behavior (dynamic analysis) or structure (static analysis). Two samples are assigned to the same family if they have a similar structure or exhibit similar behavior [13]. Classification, on the other hand, labels unknown malware samples clean or malicious [85]. A classification model is trained on labeled data that describes the behavior and structure of both benign and malicious samples. The trained model can classify previously unseen samples based on the features used to train the model [85]. Classification can also complement clustering techniques by assigning new malware samples into malware families clustering has revealed [13].

Automated malware analysis techniques provide two key benefits [13]. First, clustering allows human experts analyze entire classes of malware at once. Experts can derive signatures and develop mitigation techniques to block an entire class of malware [13]. Second, classification techniques can automatically classify and identify new malware samples that resemble existing malware. This allows experts to focus their attention to analyzing novel malware samples [13, 26].

2.2.2 Intrusion Detection

The goal of malware detection is to detect and block malicious software before it can compromise a system. However, malware detection solutions are not perfect: they might misclassify a malicious sample as clean. Additionally, malware is not the only threat to computer systems. Legitimate users might abuse or might be tricked to abuse their privileges in a computer system.

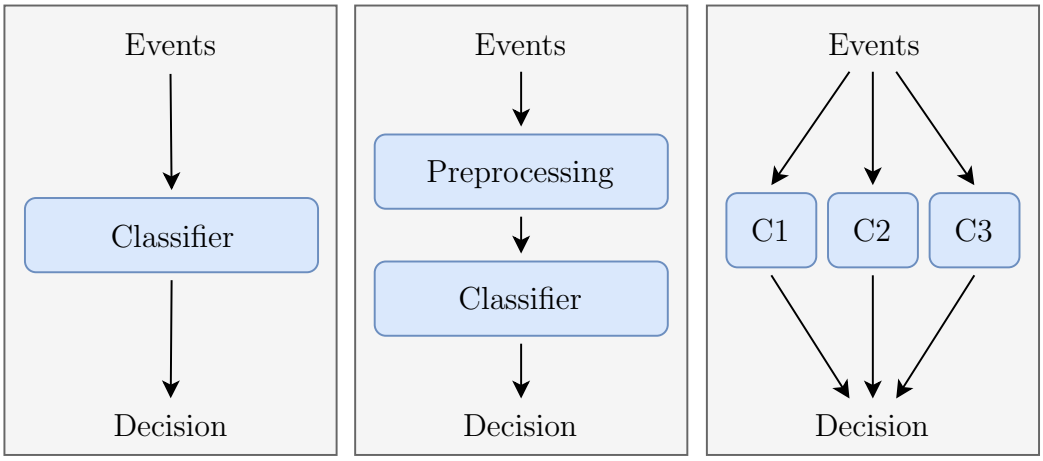
Since it is not possible to prevent all attacks, it should at least be possible to detect that an attack occurred.

Intrusion Detection Basics

Intrusion detection is defined as the problem of *"identifying unauthorized use, misuse, and abuse of computer systems by both system insiders and external penetrators"* [52, 72]. An intrusion detection system (IDS) monitors computer systems for abuse. An IDS monitors computer system activity, analyzes the collected data and alerts the system administrators of potential intrusions. The goal of an IDS is to detect intrusions, not to prevent them.

Intrusion detection systems are characterized by both the placement and the intrusion detection method. An IDS can be a host-based system or a network-based system. A host-based IDS monitors intrusions at the endpoints [24]. They monitor endpoint activity such as login attempts, executed programs and file accesses. On the other hand, a network-based IDS monitor intrusions at the network level [52]. They monitor computer network activity such as connection attempts, established connections and network data flows. Furthermore, an IDS may combine host-based monitoring with network-based monitoring to improve the overall coverage of an IDS solution [72].

Intrusion detection systems use similar detection techniques as malware detection systems. An IDS can be a misuse-based system or an anomaly-based system. A misuse-based IDS uses patterns of known intrusions to detect abuse [16]. Misuse-based systems detect known attacks and abusive behavior effectively but are unable to detect previously unseen attacks. An anomaly-based IDS compares system behavior against a normal baseline. They issue an alert if the observed behavior deviates from the baseline [16]. Anomaly-based systems can detect previously unseen attacks but are prone to generate false positives. Similarly, an IDS may combine both misuse-based and anomaly-based detection methods to improve detection coverage [25].



(a) Single Classifier (b) Hybrid Classifier (c) Ensemble Classifier

Figure 2.2: Classifier design techniques used in intrusion detection systems.

Machine Learning and Intrusion Detection

Intrusion detection systems employ techniques that are like those used in malware detection systems. Hence, machine learning can provide similar advantages to intrusion detection systems as well. For example, an IDS can train a classification model to automatically classify incoming events as benign or malicious. However, the literature on intrusion detection focuses on applying machine learning techniques to detect anomalies [73]. The research community has proposed various machine learning techniques to solve the anomaly detection problem. Common techniques include k-means clustering, support vector machines and neural networks [81].

Intrusion detection systems can be categorized by classifier design [2, 81]:

1. Single classifier systems train a single strong classifier to detect intrusions [81].

2. Hybrid classifier systems combine feature selection and reduction techniques with a strong classifier [2]. First, the incoming events are fed to a feature engineering stage [60]. In this stage, the incoming events are preprocessed with one or more feature engineering techniques. Next,

the output is fed to a strong classifier. The classifier makes a decision based on the preprocessed input [60].

3. Ensemble classifier systems combine multiple weak classifiers improve accuracy [81]. These systems train multiple models that are based on different algorithms and input features. When a new event arrives, the system feeds the event to all classifiers. Finally, the system combines the output of these classifiers to produce the final classification [53].

Applying machine learning to intrusion detection can be challenging. Sommer and Paxson [73] argue that while intrusion detection with machine learning is common in literature, machine learning techniques are not used in real-world intrusion detection systems. They highlight five key reasons for this: 1) high cost of errors, 2) lack of quality training data, 3) difficulty of turning results into operational insights, 4) variability in input data, and 5) difficulty of evaluating results. Sommer and Paxson conclude that machine learning solutions should provide sufficient insight into the capabilities and limitations of such techniques from an operational point of view [73].

2.2.3 Endpoint Detection and Response

Endpoint Detection and Response (or EDR) is an industry term used to describe a new class of tools focused on detecting and investigating suspicious activities at endpoints [19]. The goal of EDR solutions is to help organizations detect security incidents and respond to them in a timely fashion. These solutions are closely related to traditional host-based intrusion detection systems. An EDR solution typically consists of three main components: 1) an agent that collects behavioral data from an endpoint, 2) a system that analyses agent submissions in real-time, and 3) a central storage with an interface for exploratory analysis, incident response and threat hunting [20]. The biggest difference between an IDS and an EDR solution is in their response capabilities. While IDS solutions are mostly concerned with detecting incidents, EDR solutions also emphasize the capability to respond to

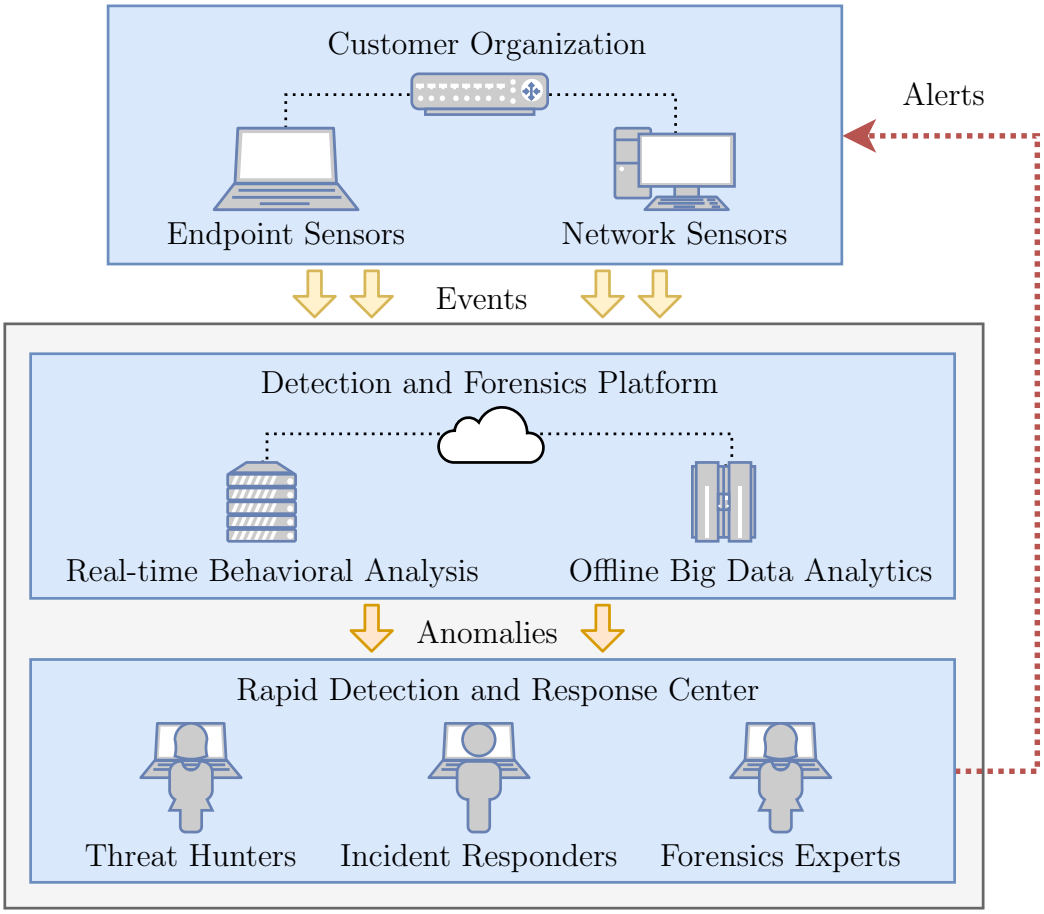


Figure 2.3: Architecture of the F-Secure Rapid Detection Service.

incidents properly.

F-Secure Rapid Detection & Response Service

The F-Secure Rapid Detection and Response Service (RDS) [29] is a managed EDR service. In RDS, lightweight sensors collect behavioral data from customer network and endpoint devices. A backend analyzes the data in real time with a combination of machine learning, statistical analysis and expert systems. If the backend detects an anomaly, it sends an alert to the Rapid Detection & Response Center (RDC). The RDC experts review all alerts, inform the customer about real incidents and provide instructions for

a response [29].

Currently, all analysis is performed in the RDS detection and forensics platform (see Figure 2.3). The real-time behavioral analysis system combines machine learning and statistical analysis with rule-based expert systems [29]. The analysis system consists of multiple layers. The first layer analyses a stream of events and flags suspicious processes for detailed monitoring. In the monitoring stage, additional information on the process behavior is collected. Finally, the alerts generated by the detailed monitoring are fed into an alert aggregation engine. The engine analyses alerts to determine if the observed behavior was hostile or not. A detection is generated if the overall behavior is deemed hostile [28]. The RDC experts analyze each detection and inform the customer of a potential incident.

The second key component of the RDS detection and forensics platform is the offline analysis platform. The offline analysis platform allows F-Secure experts to mine the collected data for signs of attacks [29]. The collected data can, for example, be used to detect new attacks retroactively or collect information on how an attack occurred to further improve detection capabilities. Compared to the real-time analysis flow, offline analysis can consider a broader context. Additionally, all machine learning models F-Secure RDS uses are trained using the data that is available via the offline analysis platform.

The centralized architecture has served F-Secure RDS well. Having all the processing logic in a centralized system has enabled fast iteration cycles on developing new detection capabilities. However, such centralized architecture poses challenges to scalability:

1. Processing large amounts of behavioral events requires considerable computational resources. The vast majority of the events F-Secure RDS collects are benign and are not part of an attack. Processing all the events centrally in the cloud is expensive given the real time processing and data retention requirements.

2. Training and managing per-sensor machine learning models is increas-

ingly expensive and difficult as the number of active sensors grows. This is because each sensor specific model needs to be trained separately with the data that sensor has generated.

These challenges are manageable with the current F-Secure RDS solution. However, F-Secure has announced a new EDR solution, F-Secure Rapid Detection & Response (RDR) service [27], that faces similar challenges at larger scale. F-Secure RDR is a partner or self-managed EDR solution targeted at small and medium-sized businesses. This means the potential customer base of the F-Secure RDR is larger than that of the F-Secure RDS which is targeted for high-end enterprise customers. The F-Secure RDS and F-Secure RDR solutions share the same analysis backend. Hence, it is crucial to keep the costs of the backend at an acceptable level to make F-Secure RDR profitable.

This concludes our discussion into context this work is performed in. While we emphasize the context throughout the thesis, the solutions we propose are by no means limited to this field. Instead, they should be applicable to all similar setups as well. The next chapter describes the motivation, requirements and goals of the simulation platform in more detail.

Chapter 3

Simulation Platform

This chapter discusses the simulation platform in more detail.

3.1 Motivation

Development of distributed machine learning solutions is not a trivial task. Most current machine learning solutions are centralized: the models are developed, trained and used in systems that run in data centers or in the cloud. Development of centralized solutions is easier in many regards: 1) all the data is readily available, 2) feedback loop during development is short, and 3) tools have been designed for such environments.

A key reason that slows the adoption of distributed learning techniques is a lack of proper development tools. Essentially, the development of a distributed solution requires a distributed development environment. A naive solution would be to use the target environment to develop distributed solutions. This approach has a few drawbacks. First, the feedback loop during development is long. Once an update to the logic is made available, it can take days for a representative sample of clients to take that into use. Second, the solution must be implemented on the target platform. This shifts focus from the development of distributed methods to platform specific implementation details.

An alternative solution is to build a distributed development environment in a data center or in the cloud. The development environment would contain a group of independent, interconnected agents that emulate the behavior of the agents of the target environment. The primary advantage of this solution is that developers can easily modify and orchestrate these agents to perform operations in the desired manner. This allows the developers to make experiments and repeat them in a reproducible fashion. Furthermore, if the primary goal is to develop distributed methods, the exact implementation of the development environment agents is not important. Hence, the agents and the logic they execute can be implemented in any technology the developers are familiar with.

3.2 Definition

The simulation platform is a development environment for distributed machine learning solutions. The goal of the platform is to simplify the development of these solutions by providing easy-to-use tools for simulating the behavior of distributed agents. This requires three key capabilities. First, the input data should be stored in an easy to use format. The data must be accessible in a per-agent time-order basis. Second, the platform should provide tools that implement common building blocks for simulations. Finally, the platform should provide a clustered compute environment for executing these simulations.

3.3 Use Cases

In our context, we have identified two important use cases that involve simulation of on-sensor functionality using data collected to a central repository:

UC1: *As a data scientist, I need an easy-to-use system to develop my ideas for new on-sensor logic and distributed machine learning capabilities.*

UC2: *As a quality engineer, I need to be able to verify that an update to the on-sensor logic does not disrupt sensor stability, cause a negative performance impact or reduce detection coverage.*

The primary concern of UC1 (or the *data science use case*) is to create capabilities that are required to develop on-sensor logic. This logic can be anything between simple rules and complex distributed machine learning techniques. The data scientists need a platform on which they can quickly prototype their ideas. They need 1) easy access to the data that would be available on the sensors, 2) the tools to emulate sensor behavior as they process this data, and 3) means to measure and evaluate the effectiveness of the solution. The goal of the data scientists is to develop new techniques that could be implemented on the sensors. The details on the sensor implementation of such logic is not relevant at this stage.

The second use case (UC2 or the *quality assurance use case*) focuses on the verification of the sensor implementation of distributed logic. Changes to implementation should be tested in a simulation environment before the changes are released to customers. For this use case the platform should 1) spin up real sensors in the cloud, 2) inject previously collected data to the sensors for processing, and 3) verify the sensors and the entire detection pipeline works as intended. While UC1 focuses on ad-hoc experimentation and development of logic, UC2 focuses on continuous testing of the actual implementation.

This work focuses on building a platform that fulfills the requirements of UC1. We included the description of UC2 as an example on how the platform could be extended in the future. We take the UC2 requirements into account on a level that our design decisions do not actively prevent UC2 from being implemented. We will not consider the requirements of UC2 to greater extent. Finally, it should be noted that while this work heavily emphasizes the context in which this platform is built, the final solution should be applicable to any similar system where independent entities process time-ordered event data.

3.4 Requirements

The functional requirements for the simulation platform are as follows:

1. The platform must support iterative development of on-sensor model training methods (local learning).
2. The platform must support development of federated learning methods that include a central server component and multiple clients talking to the server.
3. The platform must support development of peer-to-peer learning techniques that include multiple clients communicating directly with one another.
4. The platform must scale up to 10,000 sensors that perform local learning, federated learning or peer-to-peer learning concurrently.
5. The platform must provide necessary abstractions for data access and running simulations that involve one of the three distributed machine learning techniques presented earlier.
6. The platform must support Python and common Python data science libraries such as NumPy [58], scikit-learn [61] and Pandas [48].
7. The platform should provide unfiltered event data in a format that is close to the on-sensor format.
8. The platform must be implemented on top of the Amazon Web Services (AWS) public cloud and use AWS resources in a cost-efficient manner.

3.5 Evaluation Criteria

In addition to fulfilling the functional requirements enumerated above, we evaluate the simulation platform on two additional aspects: scalability and usability.

Scalability

The scalability of the simulation platform is evaluated on two dimensions: the maximum size and cost of a simulation.

- The size of a simulation is measured by the number of sensors that participate to the simulation. The platform should support local, federated and peer-to-peer learning simulations with 10,000 sensors. Scalability beyond that will be considered a bonus.
- The cost of a simulation will be measured by the cost of AWS resources per simulation. The baseline for the metric is the cost of executing simulated sensors in separate virtual machines. The cost of the platform should be fraction of the baseline costs (between 1–10 % of the baseline cost).

Usability

The evaluation of usability focuses on the developer interfaces the platform provides. It covers two dimensions: simplicity and ease of use.

- The simplicity of the platform is measured by the number of lines of boilerplate code required to implement and execute local, federated and peer-to-peer learning simulations. The business logic of the simulation does not count towards the measurement. The lower the number the better.
- The ease of use of the platform is measured by user testing. We will give users a task, observe their behavior as they fulfill this task and discuss the usability aspects of the platform with them.

Users will have an onboarding session before using the platform for the first time. That is, they do not need to be able to use the platform without an introduction to its capabilities. This is considered during evaluation.

Chapter 4

Technology Evaluation

This chapter evaluates technologies that could power the simulation platform. We limit the evaluation to technologies that provide means to schedule distributed compute tasks on a cluster of compute nodes. The evaluation proceeds as follows. First, we specify the evaluation criteria for the technology candidates. Next, we introduce five candidates in more detail. Finally, we evaluate the candidates according to the criteria and select the one that is the most suitable for simulation platform.

4.1 Evaluation Criteria

We use the following criteria to evaluate the candidate technologies:

1. **API**
    - **Description**: The technology should provide an API for scheduling distributed tasks implemented in Python.
    - **Rationale**: F-Secure data scientists use Python extensively and are familiar with Python based tooling.
2. **Compatibility**

- **Description**: The technology should be compatible with common Python data science libraries such as NumPy [58], scikit-learn [61] and Pandas [48].
- **Rationale**: F-Secure data scientists are familiar with these libraries and use them extensively in existing solutions.

3. **Flexibility**

   - **Description**: The technology must support simulations that use any of the previously discussed distributed machine learning architectures.
   - **Rationale**: The technology should not limit the solutions space by enforcing constraints on simulation complexity.

4. **AWS Integration**

   - **Description**: The technology should integrate with AWS services such as Elastic Compute Cloud (EC2) or Elastic Map Reduce (EMR).
   - **Rationale**: The platform will be built on top of AWS. Direct integration to existing AWS services reduces the amount of infrastructure setup and maintenance.

5. **Elasticity**

   - **Description**: The technology should support scaling of compute resources at runtime. That is, it should be possible to add or remove compute nodes while the system is running tasks without disrupting the process.
   - **Rationale**: Automatic scaling can help reduce costs if the task cannot fully utilize the available compute resources. It can also speed up simulations if the technology can take additional compute capacity into use while a simulation is running.

6. **License**

   - **Description**: The technology should have a permissive open source license such as Apache 2, BSD or MIT license.
   - **Rationale**: Legal concerns limit the usage of software that uses a non-permissive license.

7. **Activity**

   - **Description**: The technology must be actively developed and maintained. In essence, the community should be developing new features, fixing bugs and making new releases regularly.
   - **Rationale**: Using unmaintained technology imposes extra burden on the platform developers as they have to implement new features and fix bugs themselves. These technologies should be avoided as such maintenance is not a core competence of F-Secure.

8. **Popularity**

   - **Description**: The technology must be popular among the open source community. This can be measured with the number of stars the project repository has in GitHub.
   - **Rationale**: Using a well-known technology reduces the probability that the technology would be abandoned by the maintainer community. This can also simplify onboarding of new users as they might be familiar with the technology already.

4.2 Candidate Technologies

We chose five technologies for evaluation. The chosen technologies are: Apache Spark [88], Ciel [55], Dask [68], Dryad [40] and Ray [50]. The selection was made based on a review of publications in the field of distributed scheduling and distributed computing. The selected candidates were commonly cited in the reviewed literature.

Apache Spark

Apache Spark is a unified analytics engine for distributed data processing [88]. The Spark programming model is based on the MapReduce [23] paradigm. Like MapReduce, Spark expresses computation as a directed graph of operators that are applied to a partitioned input. In the traditional MapReduce model, each worker process reads a chunk of the input data from disk, applies the specified operator on the chunk of data and writes the results back out to disk. Spark extended this model with a notion of a resilient distributed dataset (RDD). An RDD is a collection of read-only objects partitioned over a cluster of machines [88]. The contents of an RDD can be cached in-memory and rebuilt if the node holding a partition crashes. This gave Spark a unique advantage over previous MapReduce implementations that always stored and loaded data from disk [88].

Since its initial release in 2010, Spark has introduced four built-in higher-level programming interfaces:

1. The GraphX library which implements distributed graph processing capabilities on top of Spark RDDs [84].

2. The Spark MLlib library that implements distributed machine learning on top of Spark primitives [49].

3. The Spark SQL API that provides a relational programming model for Spark applications via the DataFrame and SQL APIs [12].

4. The Spark Streaming engine that provides incremental stream processing capabilities for Spark [89].

The Spark core handles task scheduling and execution of Spark applications. A Spark application consists of a driver process and worker processes [88]. The driver process orchestrates the execution of tasks on the worker processes. The workers execute tasks the driver process schedules. Spark handles worker failures through lineage and recomputation. Spark tracks how an

RDD has been constructed. If a worker holding a partition of an RDD fails, Spark uses this lineage information to reconstruct the RDD [88]. Additionally, Spark provides a large collection of built-in data processing operations as part of its standard library and the Spark SQL API [12].

The development of Spark started at the AMPLab of University of California, Berkeley in 2009. In 2013, the Spark project was donated to the Apache Software Foundation and it became a top-level Apache project in 2014 [78]. The Apache Spark project has an active community of over 1,000 contributors from companies such as Databricks, Facebook and Google [11, 90]. The project is used in more than 1,000 organizations across various industries [90]. Additionally, AWS provides a managed Spark offering as part of their Amazon EMR service [6].

CIEL

Ciel is an execution engine for distributed computing [55]. Similar to other systems, Ciel coordinates the execution of data-parallel tasks arranged to a task graph. That is, Ciel triggers the execution of a task once the processing of input nodes has finished. The tasks at each stage are distributed to a cluster of machines. However, Ciel expands the traditional data flow model by allowing dynamic modifications of the task graph at runtime. That is, Ciel allows running tasks to add new tasks to the task graph. This capability simplifies implementation of iterative and recursive algorithms [55].

Ciel uses a custom scripting language called Skywriting to construct task graphs [55]. A Skywriting script can create new tasks, evaluate task outputs and perform unbounded iteration to build task graphs [54]. Ciel has a generic *executor* model where tasks can be implemented in languages the executor runtimes support. The Ciel project provides executor runtimes for Java, .NET, Skywriting, shell-scripts and native code [55].

Like other similar systems, Ciel provides transparent fault tolerance via re-execution of failed tasks [55]. If a worker node crashes, Ciel will automatically reassign failed tasks to be executed by another worker. If the input of

the failed tasks was lost as well, Ciel will re-execute parent tasks recursively until all of the inputs have been recovered [55].

Ciel was initially developed by the researches of the University of Cambridge Computer Laboratory in 2010 and 2011 [55]. The project saw very little interest outside the academic community and has not had any further development since 2012 [34].

Dask

Dask is a library for parallel computing in Python [68]. It provides two key features. First, Dask implements dynamic scheduling of arbitrary Python callables. The Dask library encodes callables into a task graph that defines the dependencies between tasks. A task graph is executed by a scheduler process. The Dask library provides schedulers for single-threaded, multi-threaded, multi-process and distributed execution. All built-in schedulers operate dynamically, meaning they determine the task execution order at runtime. To choose which task to run next, the schedulers use a last in, first out policy. That is, they choose tasks whose dependencies were most recently fulfilled. This strategy limits the amount of time intermediate results need to be kept in memory [68].

Second, Dask provides parallel implementations of various important data structures such as NumPy arrays and Pandas DataFrames [68]. The implementation of these high-level constructs is based on blocked algorithms. A Dask array, for example, is a combination of smaller NumPy arrays. Dask implements a subset of the NumPy array methods. The Dask implementation of these methods define a Dask task graph that parallelizes the processing of the smaller NumPy array chunks and combines partial outputs into the result. This allows Dask to process datasets that do not fit into memory [68].

The Dask project is under active development and is popular among the Python data science community. According to the Dask project statistics in GitHub [32], the project has over 4,000 stars and 200 contributors. The project also receives a steady stream of changes and makes releases at regular

intervals. Additionally, Dask supports Hadoop YARN as a deployment target [22]. The Amazon EMR service [6] provides a managed Hadoop YARN environment to which Dask applications can be deployed to.

Dryad

Dryad is a general-purpose, high performance distributed execution engine [40]. A Dryad application defines a data flow graph that includes computational vertices and communication channels between them. Each computational vertex is a simple, sequential application. Dryad parallelizes computation by executing multiple vertices simultaneously on multiple computers, or on multiple cores within a computer. The edges between vertices denote dependencies between computational vertices. The Dryad runtime handles scheduling, fault tolerance and data transfer between vertices [40].

Dryad exposes a C++ API to application developers [40]. The API is used to define the data flow graph of a Dryad application. The API also includes base classes for computational vertices. The vertices must be implemented in C++, or a C++ wrapper for a secondary language must be used [40]. In addition to the low-level C++ APIs, the Dryad ecosystem includes a DryadLINQ project [87]. The DryadLINQ project provides a higher-level, .NET API to define data transformations. The DryadLINQ system automatically compiles these high-level data processing directives to Dryad application vertices [87].

The Dryad project was originally published by Microsoft in 2006. However, Microsoft discontinued the development of Dryad in 2011 in favor of the Apache Hadoop platform [30]. The latest release of Dryad project is from November 2014 and it has not been developed since [33].

Ray

Ray is a general-purpose cluster computing framework [50]. Ray provides a unified interface for expressing both *task-parallel* and *actor-based* computation. *Tasks* are stateless functions executed over a cluster of computers.

Tasks accept input, process it and produce some output without carrying state between tasks. *Actors*, on the other hand, support stateful computation. Actors can keep actor-local state and use that to process remote function calls. The two abstractions share a common distributed execution engine that uses a dynamic task graph computation model to process distributed tasks [50].

Ray applications are implemented in Python [50]. A Ray application consists of remote functions (tasks) and remote classes (actors). These remote functions and classes are implemented as standard Python functions and classes. Remote functions must be stateless and side-effect free. When invoked, a remote function returns a *future*. The result of a future can either be retrieved via the Ray APIs or passed as argument to other remote functions. Similarly, actor classes expose methods that can be called remotely. In addition to the method parameters, the remote actor methods have access to local state of the actor class instance [50].

Ray has two key features that make it extremely flexible. First, remote functions can trigger other remote functions [50]. That is, the task graph can be extended while tasks are already running. Second, actor references can be passed to other actors and remote functions [50]. Actor references enable, for example, a direct actor-to-actor communication. This is a key feature that makes Ray suitable for various distributed applications.

The Ray project is developed at the RISELab of University of California, Berkeley. The GitHub statistics for the Ray project indicate that the project has an active developer community (over 100 contributors) and it is popular in the industry (over 6,000 stars) [35]. Additionally, Ray provides direct AWS integration for launching an auto-scaling cluster to Amazon EC2 [3, 79].

4.3 Evaluation & Comparison

Table 4.1 presents a summary of how well the evaluated technologies conform to the evaluation criteria. We discarded Ciel and Dryad from the final

Criteria	Spark	Ciel	Dask	Dryad	Ray
API	✔	✗	✔	✗	✔
Compatibility	✔	✗	✔	✗	✔
Flexibility	✔	?	✔	?	✔
AWS Support	✔	✗	✔	✗	✔
Elasticity	✔	?	✔	?	✔
License	✔	✔	✔	✔	✔
Activity	✔	✗	✔	✗	✔
Popularity	✔	✗	✔	✗	✔

Table 4.1: A summary of evaluated technologies and their conformance to the evaluation criteria. Legend: ✔ = passes the criteria; ✔ = passes the criteria with caveats; ✗ = does not pass the criteria; and ? = required information not available.

comparison as these technologies are not maintained, used or compatible with the current data analysis tooling at F-Secure. The three remaining technologies, Spark, Dask and Ray, are evaluated in more detail. The detailed evaluation focuses on the following criteria: API, compatibility and flexibility.

Apache Spark

- **API**: Spark provides Java, Python and Scala APIs for application developers. Data processing logic can also be expressed in SQL. Spark also has an extensive standard library that implements various common data processing methods.
- **Compatibility**: Spark does not integrate with Python data science libraries directly. The user can use these libraries from custom user defined functions implemented in Python, but they cannot be combined with the functionality Spark provides.
- **Flexibility**: Spark supports block-parallel computation. This means that the user can instruct Spark to execute a set of tasks and wait

for their results. Tasks cannot spawn additional tasks or communicate directly with one another. This limits the applicability of Spark for federated or peer-to-peer learning simulations. All the running tasks must finish before the worker state can be transferred to other workers via the driver process.

Dask

- **API**: Dask provides a Python API that includes distributed implementations of the NumPy array and Pandas DataFrame interfaces. The library also exposes a lower-level API for scheduling arbitrary Python functions for distributed execution.
- **Compatibility**: Dask is fully compatible with Python ecosystem including external libraries whose data structures can be serialized with the pickle library [10].
- **Flexibility**: Dask supports task-parallel computation. This means the user can instruct Dask to execute a set of tasks on remote workers. Dask supports the actor model as well [9]. However, Dask actors are experimental and lack proper cross-actor communication. This limits their usefulness for federated or peer-to-peer learning simulations.

Ray

- **API**: Ray provides a simple Python API for defining tasks and actors, calling remote functions and defining data dependencies between tasks.
- **Compatibility**: Ray is fully compatible with the Python ecosystem including external libraries whose data structures can be serialized with the pickle library [80].
- **Flexibility**: Ray provides both task-parallel and actor-based computation models. Ray is the only alternative to support the direct actor-to-actor communication pattern.

4.4 Final Selection

Based on the previous comparison, Ray is the most suitable technology for the simulation platform. There are two key factors that favor this selection:

1. Ray has been designed for simulating complex reinforcement learning environments. It provides a flexible actor model in which actors can communicate directly with one another. The direct cross-actor communication model simplifies the implementation of federated and peer-to-peer machine learning simulations. The alternatives do not support such cross-actor communication pattern.

2. Ray integrates to Amazon EC2 directly. It also provides snappy auto-scaling that reacts to changes in load in seconds. The alternatives integrate to AWS via the Amazon EMR service. Amazon EMR requires extra infrastructure and has slower auto-scaling implementation. The F-Secure experience on Amazon EMR is that auto-scaling takes up to 5 minutes to detect the need to scale and then up to 15 minutes for the new nodes to be available.

Therefore, the simulation platform will be built on top of the Ray framework. The next chapter describes how the platform is implemented on top of Ray and AWS.

Chapter 5

Implementation

The previous chapters described the simulation platform requirements and technologies that could be used to implement the platform. This chapter discusses the simulation platform implementation on Amazon Web Services (AWS) and the Ray framework.

5.1 Input Data

The platform collects input data from the F-Secure RDS event processing pipeline. It hooks into the pipeline to make a copy of all incoming data from a subset of endpoint sensors. The sensors from which data is collected are chosen at random. We can change the size of the sample i.e. the number of sensors we collect data from at will. The data is stored to Amazon S3 [7] in a compressed JSON format.

The Amazon S3 storage structure is optimized for the expected access patterns. The storage structure design is based on two key observations. First, while the system simulates the behavior of multiple sensors, each simulated sensor uses the data of a single real sensor as input. Therefore, it must be easy to read data for one sensor only. Second, simulations process events in the order they occurred on real sensors. Furthermore, the length of the time period simulations need data from can vary. Hence, it must be

possible to limit the input to a specific time range as well.

Amazon S3 operating principles put additional constraints on the storage structure. Amazon S3 is an object store that stores blobs of data (*objects*) in buckets (collection of *objects*). Each object has a unique key that must be supplied to access the object. The keys of objects that belong to a specific bucket can be listed through the Amazon S3 APIs. However, listing a bucket is slow if the bucket contains large number of objects.

We can speed up bucket listing by storing objects in a *directory* structure. Two objects belong to the same *directory* if their keys share a common prefix. A delimiter character (by default a forward slash) splits a prefix into a directory hierarchy. The Amazon S3 API allows a list operation to be restricted to objects and directories that belong to a specific directory. Listing a directory is faster than listing the entire bucket. Additionally, Amazon S3 can be configured to deliver a full object listing for a bucket as an inventory file. Amazon S3 updates the inventory at most once a day which means it does not reflect recently created or removed objects.

Based on the previous observations, we store the raw submission data in the following structure:

`_sensor_id=<sensor_id>/_date=<YYYY-MM-DD>/<uuid>.json.gz`

This structure allows the platform to 1) list all sensor IDs i.e. list unique directories under the root directory, 2) list all submissions of a sensor i.e. list objects under a given `_sensor_id` prefix recursively, and 3) list all submissions of a sensor for given day i.e. list objects under a given `_sensor_id` and `_date`.

5.2 Simulation Clusters

The Ray framework processes distributed tasks on a cluster of computers. A Ray compute cluster consists of one head node and multiple worker nodes. The cluster nodes can be provisioned manually or with one of the cluster providers built into Ray. We use the Ray cluster provider for AWS to provi-

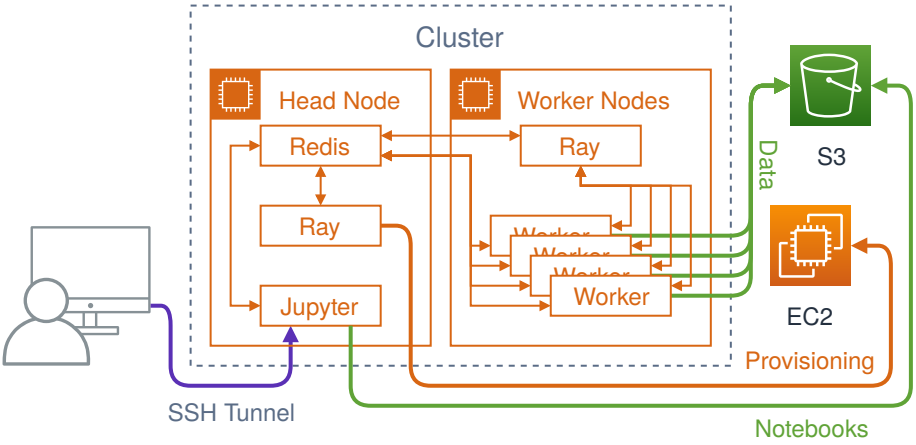


Figure 5.1: Ray cluster architecture and connectivity between internal and external components. Note: The Ray component included in this figure consists of multiple independent services. The figure combines these services into a single component for simplicity.

sion clusters for simulations. The AWS provider provisions the cluster head node to Amazon EC2 first [79]. Once started, a Ray process running on the head node starts to manage the worker nodes of the cluster. Ray launches and terminates EC2 instances automatically to adjust the worker capacity to current requirements [79].

Figure 5.1 presents the simulation cluster architecture. Ray runs a set of processes on each cluster node. The head node hosts both a set of Ray services and a Redis server. The Ray daemon services monitor the cluster state and manage worker instances through the EC2 API. The Redis server is a control plane Ray uses to store global state and distribute tasks to worker nodes [50]. The worker nodes execute a Ray daemon services as well. On a worker node, the Ray daemon services monitor and manage a pool of worker processes that execute tasks. The workers fetch tasks from the Redis server running on the head node.

5.2.1 Cluster Management

Users manage clusters through a command line interface (CLI). The CLI consists of two tools: a custom `simulation-cli` command and the `ray` CLI tool. The `simulation-cli` command allows users to generate a configuration file for a personal Ray cluster. The tool renders configuration template with user and environment specific details. The template enforces best practices on cluster auto-scaling, instance configuration and resource tagging. The generated configuration file can be used with the `ray` tool to manage a Ray cluster.

The user workflow for managing a cluster goes as follows:

1. User generates a cluster configuration file with the `simulation-cli` command:

   ```
   simulation-cli cluster-config --contact user@example.com
   ```

   This command produces a `cluster-config.yaml` file with the cluster configuration.

2. User launches a cluster with the Ray CLI:

   ```
   ray up cluster-config.yaml
   ```

3. Once running, the user can execute code on the cluster with the `ray exec` command. They can, for example, start a Jupyter Notebook [37] on the head node and forward the Jupyter Notebook port over an SSH tunnel to the user machine:

   ```
   ray exec cluster-config.yaml --port-forward=8899 \
     'jupyter notebook --port=8899'
   ```

4. When the user is done, they terminate the cluster with the Ray CLI:

   ```
   ray down cluster-config.yaml
   ```

We provide a custom Python package that includes both tools discussed in this section. The package can be installed from an internal Python package repository with the Python Package Installer (pip) [66] tool.

5.2.2 Cost Management

We use Amazon EC2 Spot Instances [5] to reduce simulation costs. Amazon EC2 Spot Instances provide up to 90 % discount on unused EC2 capacity compared to standard EC2 instances. However, AWS may reclaim spot instances if they are out of EC2 capacity. Such interruptions do not pose a problem for the simulation platform as 1) interruptions are rare and 2) Ray can recover from node failures.

Additionally, the default auto-scaling configuration is extremely aggressive to scale down the worker capacity when the cluster is idle. We also employ other F-Secure internal tools to detect underutilized EC2 instances and notify the instance owners about them.

5.2.3 User Interface

We use Jupyter notebooks as the primary user interface to the simulation platform. Jupyter Notebook [37, 43] (originally called iPython [62]) is a web application that allows users to create documents (*notebooks*) with executable code, text and visualizations. Notebook content, including executable code, can be modified interactively through the browser. Notebook code is executed on the machine Jupyter is running on. Hence we run the Jupyter Notebook application on the cluster head node.

5.3 AWS Infrastructure

The simulation platform requires infrastructure to run on. Figure 5.2 shows a high-level overview of the AWS infrastructure of the simulation platform. We describe the infrastructure in more detail next.

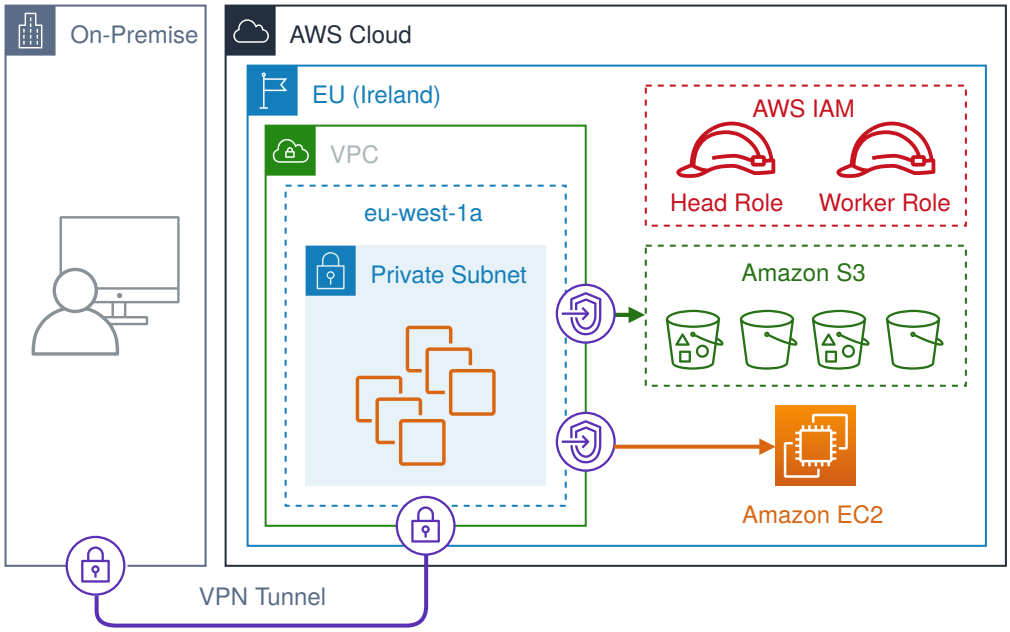


Figure 5.2: The AWS infrastructure the simulation platform runs on.

Amazon S3 Buckets

We provide four S3 buckets to store data: 1) A raw submission bucket for raw submission data, 2) a simulation data bucket for inflated event data, 3) a workspace bucket for temporary results and simulation outputs, and 4) a notebook bucket for Jupyter [37, 43] notebooks. The buckets have a bucket policy that deny access from outside the simulation platform. We also utilize Amazon S3 Object Lifecycle Management to delete collected data after 30 days. The 30 day retention policy is required for cost management, compliance and privacy reasons.

Amazon Virtual Private Cloud (VPC) Subnets

Cluster instances run in a single Availability Zone (AZ) of the EU (Ireland) region. Availability Zones are isolated locations (i.e. data centers) within an AWS region that provide fault isolation [8]. A single-AZ topology reduces cross-node communication latency and eliminates cross-AZ traffic costs. If an

AZ fails, we can move simulation clusters to a secondary AZ. Furthermore, the clusters run in an isolated subnet i.e. they do not have a direct route to the internet. Lack of internet access increases the difficulty of data exfiltration. Access to Amazon S3 and Amazon EC2 are provided via a VPC endpoint and a proxy service.

AWS Identity and Access Management (IAM) Roles

The platform provides two IAM roles for the Ray compute cluster instances: the head role and the worker role. The Ray cluster head nodes use the head role to manage worker nodes. The role allows the head nodes to start and terminate EC2 instances that are tagged with a `ray-node-type: worker` tag. The `ray-node-type` is a resource tag Ray sets on the worker nodes. The resource tag restrictions prevent the head node from disrupting other EC2 workloads running on the same account. Similarly, the worker nodes use the worker role to access AWS services. Both the head and the worker roles grant the Ray cluster instances access to the S3 buckets described earlier.

5.4 Simulation Design

The Ray framework provides low-level primitives for distributed computing. A simulation must combine these primitives to a full application. We provide best-practices on the simulation application design to simplify development.

The simulation application design is based on three key observations:

- Input data should be processed in time-order. This models the behavior of real sensors which process events as soon as they occur. The order is especially important for models that detect anomalies in sequences of events.

- Input processing should be synchronized. That is, the simulation should preserve the global order of input over all the sensors. Synchronization

```python
@ray.remote
def sensor_function(chunk):
    # ... (process chunk)

for chunk in chunk_input(input, chunk_length="1 day"):
    for sensor_input in chunk:
        sensor_function.remote(sensor_input)
    wait_for_remote_tasks()
# ... (analyze results)
```

Listing 1: Pythonic pseudocode for a generic simulation application.

prevents the relative order in which events are processed on different sensors from affecting the outcome.

- Simulated sensors cannot reserve resources for the duration of a simulation. Instead, sensors must share resources with one another to improve efficiency.

To meet the requirements, we propose an iterative design for simulations. In this design, the input is split into chunks that contain data from a specific time period. Simulations process these chunks one by one in order. The length of the period can vary depending on simulation requirements. However, the data storage is optimized for reads at daily granularity. Hence, the chunk length should be at least one day.

Simulated sensor logic is implemented as a Ray remote function. The remote sensor function receives a chunk of data from a single sensor as input, processes the data and saves the results to a state object. The state object is kept in the Ray object store and passed to each invocation of the sensor function. Simulations iterate over chunks of input and trigger the remote sensor function to process the data. The simulations wait for all the remote sensor tasks to finish before continuing to the next chunk. See Listing 1 for an example of how a simulation application implements this design.

This design provides three key benefits. First, it preserves the order of input within a single sensor. Chunks are processed in time order, and the events within a chunk can be sorted if required. Second, this design preserves global order of input at the specified granularity (i.e. chunk length). Enforcing the exact order over all the simulated sensors is not feasible due to the overhead synchronization would cause. Finally, the design provides sensor functions a natural place at the end of each chunk to release resources to other sensors.

5.4.1 State Management

The remote sensor functions need to keep state between iterations. Ray provides two state management models: actors and the Ray object store.

The Ray object store stores remote function inputs, return values and user specified objects. Each object in the store has a unique handle that can be used to fetch the value. A call to a remote function returns a handle to the return value of the function. The handle to the return value can be given as an argument to other remote functions. This provides a convenient way to pass state between two stateless remote sensor function calls. We recommend this pattern for most simulations.

Actors are Python classes whose methods can be called remotely. When an actor is initialized, Ray creates an instance of the actor class on a remote node. Remote calls to actor class methods are routed to the actor class instance. The actor class instance is active for the entire lifetime of the application. Therefore, actor classes can keep state in instance variables. Actor-based state storage is recommended if a simulation requires other actor functionality such as direct sensor-to-sensor communication.

5.4.2 Technique Specific Implementation Details

Section 2.1 presented three categories of distributed machine learning techniques: local learning, federated learning and peer-to-peer learning. All three

techniques are compatible with the iterative design presented earlier. However, each technique has unique characteristics that impose additional requirements on the implementation.

Local Learning

In local learning, a sensor trains a model from locally available data. It does not use or share any information with external entities during training. Hence, we can relax the synchronization requirement of the original design for local learning. There is no need to synchronize processing if sensors do not interact with one another. Therefore, a sensor can start processing the next chunk even before other sensors have finished processing the current chunk. This improves resource utilization as the slowest sensor does not block faster sensors from progressing.

Federated Learning

In federated learning, a group of sensors collaborate to train a model. Each sensor trains a model with local data and sends model updates to a parameter server. The parameter server combines updates from individual sensors into a global model. Thus, simulations that use federated learning need a parameter server component. As recommended by the Ray authors, the parameter server should be a Ray actor. The actor should implement the necessary logic for combining model updates into a global model. Simulations should initialize one parameter server actor per simulation and pass the remote sensor functions a reference to the parameter server actor instance. The sensor functions can use the actor reference to read parameters and send updates to the global parameter server.

Peer-to-Peer Learning

In peer-to-peer learning, a group of sensors share information directly with one another to build a model from data different sensors have observed. To

enable sensor-to-sensor communication within a simulation, sensors should be implemented as Ray actors. The actor class should implement sensor processing logic, peer sensor management, and expose remote functions for cross-sensor communication. Simulations should initialize an actor instance for each sensor and distribute peer sensor handles to them. The sensor processing functions can use the peer sensor handles to communicate other sensors.

5.5 Simulation Library

The simulation platform includes a Python library that provides utilities for implementing simulations. The utilities range from high-level methods for orchestrating simulations to low-level building blocks. The functionality includes interfaces and base classes for common components, helper functions for data access and methods for orchestrating simulations. The library is available on all simulation clusters by default and can be used without additional setup.

Common Components and Base Classes

The library provides following common components and base classes:

- `ParameterServerBase` – A common base class for federated learning parameter servers. The class implements a key-value store with `get(key)` and `put(key, value)` methods. It also defines an `update(key, delta)` method for extending classes to implement simulation specific update logic.

- `P2PSensor` – A base class for peer-to-peer simulation sensors. The class implements a `set_peers(sensors)` method for configuring peer sensors. It also defines a `process(sensor_input)` method for extending classes to implement sensor logic.

- `sensor_input` – A dictionary structure used to pass S3 object coordinates to per-sensor input data. Each `sensor_input` object contains two fields. The `sensor_id` field contains a unique identifier for the sensor the input belongs to. The second field, `s3_objects`, contains a list of S3 object coordinates to the data of the sensor.

Data Access Layer

The `data` module provides two methods for working with the input data:

- `prepare_submissions(filters)` – A method that filters input, processes matching submissions and returns a list of `sensor_input` dictionaries for sensors included in the input. Currently, filtering is based on information encoded in the S3 object keys.
- `get_data(sensor_input)` – A method that takes a `sensor_input` dictionary as input and returns an iterator that yields parsed event data from the input objects.

Simulation Harness

The `harness` module provides methods for orchestrating simulations.

- `local_learning_simulation(sensor_inputs, sensor_func)` – Performs a local learning simulation using the given data as input. The second argument, `sensor_func`, is a Ray remote function that implements the sensor logic. The method splits the inputs into daily chunks and calls the sensor function to process input one chunk at a time. On each invocation, the sensor function receives two arguments: a state object and a `sensor_input` struct for the daily input of a sensor. The sensor function should process the given input and return an updated state object to the caller.
- `federated_learning_simulation(sensor_inputs, sensor_func, parameter_server)` – Performs a federated learning simulation with

the given data as input. The `parameter_server` argument should be a reference to a Ray actor class that extends `ParameterServerBase`. Like the previous case, the method splits the input into daily chunks and calls the `sensor_func` with state and input as arguments. In this case, however, the `sensor_func` also receives a reference to the parameter server actor as a third argument.

- `p2p_learning_simulation(sensor_inputs, sensor_class)` – Performs a peer-to-peer learning simulation with the given data. The `sensor_class` argument should be a reference to a Ray actor class that extends the `P2PSensor` class. The method initializes a sensor actor for each sensor in the input and uses `P2PSensor.set_peers()` method to configure peers. Like the two previous cases, the peer-to-peer simulation harness splits the input into daily chunks and invokes the `process` method of the sensor actor with the daily input as the only argument. The sensor internal state should be stored in an instance variable of the actor class.

Finally, we note that the library follows a modular design. The functionality of the methods and components discussed here usually combine smaller functions into one higher level interface. For example, the `prepare_submissions` method combines functions that discover, filter, process and cache the input data into a single method. The low-level building blocks are available for simulations that have special requirements or do not fit the design patterns discussed in this chapter.

This concludes our discussion on the platform implementation. The next chapter evaluates the platform on the criteria defined in Chapter 3.

Chapter 6

Evaluation

The previous chapter discussed the simulation platform implementation and provided best-practices for authoring simulations. In this chapter, we evaluate the implementation on the two criteria defined in Section 3.5: scalability and usability.

6.1 Evaluation Setup

We start the discussion by describing the use cases and the compute environment the evaluation was performed in.

6.1.1 Sample Use Case

To evaluate the platform, we implement simulations for a fairly simple model called *Anomalous Process Creation* (APC). APC is a statistical model that evaluates if a process launch event is anomalous. In APC, a process launch event is anomalous if 1) the child process is launched often, 2) the parent launches other processes often, and 3) the (parent, child) combination is rare i.e. the parent rarely spawns the given child. The goal of APC is to detect if a common process launches another common process in anomalous way. For example, it's common for both Microsoft Word and cmd.exe (Command Prompt) to be running but Word should never launch cmd.exe.

In practice, the APC model is a database of aggregated process launch counters. The model has counters for 1) how many times has a process launched other processes (parent counts), 2) how many times has a process been launched by other processes (child counts), and 3) how many times has a specific parent launched a specific child (parent-child counts). When a new process launch is observed, the model uses the counters for the respective processes to compute an anomaly score. Our experiments focus on building the model database – evaluation of the model performance is out of the scope of this work.

We chose to evaluate the platform with the APC model for two key reasons. First, the model is simple and lightweight. This allows us to measure the overhead and limitations of the simulation platform – not the overhead of a specific model. Second, the model can be built with all three distributed machine learning architectures. Hence, we can use the same example to evaluate simulations for all three techniques.

6.1.2 Compute Infrastructure and Dataset

The simulations are performed in AWS on a cluster of up to 11 general purpose m5.large instances. The m5.large instance provides 2 vCPU cores of Intel Xeon Platinum 8000 series (Skylake-SP) processors with clock speeds of up to 3.1 GHz [4]. The instances have 8 GB of memory and up to 10 Gbps of network bandwidth. The instances use the AWS Deep Learning AMI (Ubuntu) version 22.0 and Ray version 0.6.6. All cluster nodes run in the same availability zone of the EU Ireland region.

The input data includes process launch events from over 18,000 endpoints. The input was collected from a five-day period in the beginning of March 2019. The input is split into 77,000 objects in S3 that contain over 176 million process launch events. The events have been pre-processed to include three fields: the parent process name, the child process name and a timestamp when the event occurred. We use random samples of this dataset to test the platform at different scales.

6.2 Scalability

As discussed in Section 3.5, we measure the platform scalability on two criteria: size (number of sensors) and cost of the simulations.

6.2.1 Size of Simulations

The maximum size of simulations was tested by running simulations with different amounts of data. The goal for the platform was to support simulations with up to 10,000 sensors.

Local Learning

The platform supports local learning simulations with 10,000 sensors without issues. Furthermore, we performed a successful simulation with the full dataset of 18,000 sensors. We believe the platform could scale to larger simulations as well due to the simulation design. Ray is designed to process thousands of remote tasks with very small overhead. This is exactly how the local learning simulations use Ray. The most likely limiting factor is the total size of the sensor state which must fit the Ray in-memory object store.

Federated Learning

We were able to run a federated learning simulation with 10,000 sensors on the platform. However, we identified two issues with the parameter server design:

- The single parameter server instance causes extra overhead to simulations. We measured a 6 % increase in simulation runtime compared to a similar local learning simulation. We believe the overhead is caused by additional communication between sensors and the parameter server. In addition, the parameter server actor applies updates sequentially in a single thread while sensors send updates in parallel. This makes the parameter server a bottleneck in the simulations.

- The parameter server actor would hang when a large global model was requested from the server. This issue occurred in larger APC simulations as the size of the APC model depends heavily on the amount of data available. We were able to work around this issue by reducing the size of the model on the actor side before returning it to the requester. Simulations that produce and move large objects between remote functions need to take this limitation into account.

We can address these issues by 1) using multiple parameter server actors, or 2) using an external parameter server service. The multi-actor solution would spread model update processing to multiple actors and compute nodes. This solution removes the bottleneck of a single actor but introduces additional complexity by requiring parameter synchronization between multiple actor instances. On the other hand, the external parameter server service would decouple the simulation from parameter storage completely. The external service is independently scalable and can utilize alternative data stores to store data.

Peer-to-Peer Learning

We were able to run a peer-to-peer simulation with 1000 sensors on the platform. We discovered two limitations that prevent execution of large peer-to-peer learning simulations.

- Ray actors have a high memory overhead. Each actor instance requires a worker process to host the actor for the entire actor lifetime. If a simulation creates thousands of actors, Ray needs to create hundreds of worker processes to host all the actor instances. Each worker is a standalone Python process that requires a sizable chunk of memory to operate. Hence, if a simulation creates several thousand actors, the cluster must have a lot of memory.

- Direct actor-to-actor communication adds extra overhead to the simulation process. If actors are fully connected i.e. every actor sends

updates to every other actor, the number of update messages sent during simulation grows exponentially. For example, in a simulation with 10,000 sensors, each sensor sends updates to 9,999 sensors on each iteration. In total, the sensors make nearly 100 million remote function calls per iteration. This is slow and unrealistic. Instead, it is highly likely that sensors communicate with a small number of sensors only. Hence, simulations do not need full connectivity between sensors either.

To avoid Ray actor limitations, we can implement peer-to-peer learning simulations with remote functions and message passing. In this design, the sensor logic is implemented as a Ray remote function. When called, the remote function receives a handle to a message broker actor as argument. The message broker actor exposes methods to send and receive messages. The remote sensor function would fetch pending messages, process them and send messages to other sensors on every invocation. Message passing has higher performance as 1) the simulation does not initialize many actors, and 2) sensors can fetch and send messages in batches instead of making one function call per recipient.

6.2.2 Simulation Costs

The cost of a simulation is measured as the cost of AWS resources used during simulation. The experimental simulations were performed with 11 m5.large spot instances. The price for an m5.large spot instance in EU (Ireland) region was $0.036 per hour. Therefore, the 11 nodes have a total cost of $0.396 per hour. The previous section showed that it is possible to simulate 10,000 sensors or more on an 11 node cluster. Running 10,000 sensors as separate m5.large instances would've cost $360 per hour. Even with a smaller t2.small instance (1 vCPU and 2 GB of memory), the cost of running 10,000 sensors in separate virtual machines would've been $75 per hour ($0.0075 per hour per instance).

Moreover, using a large cluster does not necessary increase the cost of a simulation. If we use a larger cluster in a simulation, the simulation likely

completes faster than on a smaller cluster. Instead of running a small cluster for a longer time, we can use a large cluster for a shorter period. Amazon EC2 uses per second billing which means we only pay for the seconds the nodes are running. For example, if a simulation takes 60 minutes on a 10-node cluster but only 30 minutes on a 20-node cluster, the cost of simulation is the same with both clusters.

We performed an experiment with local APC model to see how the size of the cluster affects the simulation runtime. With a cluster of 11 m5.large nodes, the local APC simulation took 2 minutes and 52 seconds (average of three runs). A smaller cluster of 6 m5.large nodes completed the same simulation in 5 minutes and 17 seconds. In total, the smaller cluster used $6 \times 317 = 1902$ m5.large instance seconds while the larger cluster finished the simulation in $11 \times 172 = 1892$ instance seconds. Thus, the simulation that used the large cluster was cheaper than the one that used a smaller cluster.

6.3 Usability

We evaluate the usability of the platform on two criteria: simplicity and ease of use.

6.3.1 Simplicity

The simplicity of the is measured by the lines of code that are required to implement simulations.

Cluster Management

As shown in Section 5.2.1, the users need to use five commands to manage their clusters:

```
# Install cli tools
pip install simulation_tools[cli]
```

```
# Create cluster configuration
simulation-cli cluster-config --contact user@example.com

# Create cluster
ray up cluster-config.yaml

# Connect to cluster
ray exec cluster-config.yaml --port-forward=8899 \
    'jupyter notebook --port=8899'

# Terminate cluster
ray down cluster-config.yaml
```

We consider this to be simplest possible way to manage Ray clusters. An easier alternative would be to have a shared cluster for all users. This cluster would run constantly and scale up or down depending on the cluster load. Users would just need to have an SSH key they can attach to the cluster. This approach, however, requires the head node to be up all the time which generates unnecessary costs. Furthermore, Ray does not support fine-grained control over task scheduling. The current scheduling strategy could lead to one user hogging all the cluster resources. Hence, this idea is impractical.

Simulation Implementation

Different kinds of simulations require boilerplate as follows:

- A local learning simulation requires 8 lines of boilerplate code:

```
import ray

from simulation.lib.data import prepare_submissions
from simulation.lib.harness import local_learning_simulation

@ray.remote
def sensor_func_local(state, sensor_input):
    # Simulation specific logic
```

```
    return state

ray.init(redis_address='localhost:6379')

sensor_inputs = prepare_submissions()
final_states = \
    local_learning_simulation(sensor_inputs, sensor_func_local)
```

- A federated learning simulation requires 12 lines of boilerplate code:

```
import ray

from simulation.lib.data import prepare_submissions
from simulation.lib.federated import ParameterServerBase
from simulation.lib.harness import federated_learning_simulation

@ray.remote
class ParameterServer(ParameterServerBase):
    def update(self, key, delta):
        # Simulation specific logic
        pass

@ray.remote
def sensor_func_federated(state, sensor_input, parameter_server):
     # Simulation specific logic
    return state

ray.init(redis_address='localhost:6379')

sensor_inputs = prepare_submissions()
final_states, parameter_server = federated_learning_simulation(
    sensor_inputs, sensor_func_federated, ParameterServer)
```

- A peer-to-peer learning simulations requires 10 lines of boilerplate code:

```
import ray
```

```
from simulation.lib.data import prepare_submissions
from simulation.lib.p2p import P2PSensorBase
from simulation.lib.harness import p2p_learning_simulation

@ray.remote
class Sensor(P2PSensorBase):
    def process(self, sensor_input):
        # Simulation specific logic
        pass

ray.init(redis_address='localhost:6379')

sensor_inputs = prepare_submissions()
sensors = p2p_learning_simulation(sensor_inputs, Sensor)
```

Based on the above examples, we consider the platform successful in terms of how simple it is to implement a simulation.

6.3.2 Ease of Use

We measured the ease of use of the platform with expert interviews. We described the platform and simulation design to the experts and asked them to provide feedback on the design. We also held a hands-on session with the experts to evaluate the usability of tools and libraries created in this work.

Tooling and Technologies

Our users think the cluster management tools are intuitive and easy to use. The cluster management workflow is similar to other F-Secure internal processes. Hence, it is easy for users start up their clusters and connect to them.

Additionally, our users saw Ray as an interesting technology that could fit other use cases as well. Their primary concern was the stability of Ray. Ray allocates CPU and GPU resources to remote functions but does not enforce

these limits. Furthermore, Ray does not track other cluster resources such as disk, network or memory. A simulation can destabilize the entire cluster by overusing cluster resources. Currently, the platform does not provide means to limit or control the resource usage of simulations. Such functionality could be a useful addition to later iterations of the platform.

Simulation Architecture

The iterative simulation architecture presented earlier made sense to F-Secure experts. Their biggest concern was that a generic architecture might not fit all use cases. This is expected as it is impossible to create a generic architecture for a system whose use cases are not known at the time of creation. However, the current architecture is a good starting point and can be developed further to support new use cases.

Second concern that came up in the discussions was the chosen level of granularity (1 day). The concern was if 1 day is too long for some use cases. The granularity was chosen to provide a good balance between simulation accuracy and performance. However, some use cases utilize online learning in which the model is continuously updated as events occur. Processing data in too large chunks can reduce the accuracy of the simulations that test such models. It is possible to apply online learning principles within a single simulated sensor but synchronizing the process over multiple simulated sensors in not feasible.

Simulation Library

Our users think the simulation library provided a good baseline of functionality to help them create simulations. The discussions and hands-on testing of the library revealed three common concerns:

- The library assumes the simulation data is stored in an Amazon S3 bucket and that the objects follow our naming convention. Some simulations require generated data not stored in Amazon S3 or other

datasets that use different naming convention. Right now, users must either save the data to Amazon S3 in the correct format or access the data and implement the simulations themselves. We should make data access layer customizable to allow users to use non-standard datasets in the simulations.

- Users need access to metadata of the input data we provide. Currently, the library hides away the dataset details from the users and does not provide means to customize the datasets used for simulations. For example, users would like to get an inventory of available data to discover the endpoint sensors for which data is available. They could use this inventory to choose an appropriate sample for their simulations.

- The users were concerned with how opinionated the harness functions are. Their experience has shown that it is difficult to generalize machine learning processes which usually require a lot of context specific tweaking. They thought the harness functions should be extensible and allow users to tweak the process to their needs.

This concludes our evaluation of the simulation platform. To summarize, the platform fulfills the criteria we set out in Chapter 3. While the current implementation has limitations, they do not prevent users from performing simulations. The next and the final chapter summarizes the work.

Chapter 7

Conclusion

This work aimed to simplify development of distributed machine learning solutions. Currently, most development tools are designed for centralized solutions that train and operate models in a data center or in the cloud. In distributed learning, models are trained and used over a large number of client endpoints instead. This imposes additional requirements to development tooling as well. Essentially, distributed learning requires a distributed development environment where these solutions can be tested in.

We developed a simulation platform to simplify the development of distributed machine learning solutions. The platform enables development, testing and verification of distributed machine learning solutions in a cloud environment. It allows developers to quickly test their ideas, iterate on promising ones and validate the solutions at scale.

We made two key contributions in this work. First, we introduced a working technical implementation of the simulation platform. The platform uses the Ray distributed execution framework to simulate a distributed system on top of the AWS public cloud. Second, we developed models for simulating different distributed machine learning techniques. The techniques include local, federated and peer-to-peer learning. We also created a library to help developers implement simulations on the platform.

Our experiments show that the platform can scale to real-world use cases.

However, the platform cannot scale more complex simulations to extent we hoped for. This is something we hope to address in the future with improvements to the Ray framework and the simulation architecture. We also showed that the users find the platform and the programming model easy to use and understand. The platform has some usability issues, but we hope to address those in the future.

To summarize, the platform fulfills the expectations we set out in the beginning of this work. While the platform is not perfect, it provides the functionality required to develop distributed machine learning solutions. We hope to continue the development of the platform by improving the current solution and adding additional features for evaluating distributed techniques. The platform could, for example, provide means to measure the CPU, memory and network bandwidth requirements of a distributed solution during simulation. Another interesting direction of research is to see how the simulated logic could be transformed into the target environment. These are both important aspects in the development of distributed solutions and we hope to address them in future work.

Bibliography

[1] Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., and Zhang, L. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 308–318.

[2] Aburomman, A. A., and Reaz, M. B. I. A survey of intrusion detection systems based on ensemble and hybrid classifiers. *Computers & Security 65* (2017), 135–152.

[3] Amazon Web Services. Amazon EC2. [Online], Available: `https://aws.amazon.com/ec2/`, Accessed on: March 26, 2019.

[4] Amazon Web Services. Amazon EC2 M5 Instances. [Online], Available: `https://aws.amazon.com/ec2/instance-types/m5/`, Accessed on: May 18, 2019.

[5] Amazon Web Services. Amazon EC2 Spot Instances. [Online], Available: `https://aws.amazon.com/ec2/spot/`, Accessed on: April 10, 2019.

[6] Amazon Web Services. Amazon EMR. [Online], Available: `https://aws.amazon.com/emr/`, Accessed on: March 26, 2019.

[7] Amazon Web Services. Amazon Simple Storage Service (Amazon S3). [Online], Available: `https://aws.amazon.com/s3/`, Accessed on: April 3, 2019.

[8] Amazon Web Services. Regions and Availability Zones, User Guide for Linux Instances, Amazon Elastic Compute Cloud. [Online], Available: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`, Accessed on: April 17, 2019.

[9] Anaconda, Inc. Actors — Dask.distributed Documentation. [Online], Available: `https://distributed.dask.org/en/latest/actors.html`, Revision: 716142e8, Accessed on: March 27, 2019.

[10] Anaconda, Inc. Limitations — Dask.distributed Documentation. [Online], Available: `https://distributed.dask.org/en/latest/limitations.html`, Revision: 716142e8, Accessed on: March 27, 2019.

[11] Apache Spark. Current Committers. [Online], Available: `https://spark.apache.org/committers.html`, Accessed on: March 21, 2019.

[12] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (2015), ACM, pp. 1383–1394.

[13] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. Scalable, behavior-based malware clustering. In *NDSS* (2009), vol. 9, Citeseer, pp. 8–11.

[14] Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., and Seth, K. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 1175–1191.

[15] Bottou, L., and Bousquet, O. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems 20*, J. C. Platt,

D. Koller, Y. Singer, and S. T. Roweis, Eds. Curran Associates, Inc., 2008, pp. 161–168.

[16] Brown, D. J., Suckow, B., and Wang, T. A survey of intrusion detection systems. *Department of Computer Science, University of California, San Diego* (2002).

[17] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (2011), ACM, pp. 15–26.

[18] Chebyshev, V., Sinitsyn, F., Parinov, D., Kupreev, O., Lopatin, E., and Liskin, A. IT threat evolution Q3 2018. Statistics. *Kaspersky Lab* (Nov 2018). [Online], Available: `https://securelist.com/it-threat-evolution-q3-2018-statistics/88689/`, Accessed on: February 18, 2019.

[19] Chuvakin, A. Named: Endpoint threat detection & response. Gartner Blog Network, July 2013. [Online], Available: `https://blogs.gartner.com/anton-chuvakin/2013/07/26/named-endpoint-threat-detection-response/`, Accessed on: February 15, 2019.

[20] Chuvakin, A. On Endpoint Sensing. Gartner Blog Network, July 2013. [Online], Available: `https://blogs.gartner.com/anton-chuvakin/2013/07/03/on-endpoint-sensing/`, Accessed on: February 15, 2019.

[21] Damodaran, A., Di Troia, F., Visaggio, C. A., Austin, T. H., and Stamp, M. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques 13*, 1 (2017), 1–12.

[22] DASK-YARN AUTHORS. Dask-Yarn. [Online], Available: `https://yarn.dask.org/en/latest/`, Accessed on: March 21, 2019.

[23] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (San Francisco, CA, 2004), pp. 137–150.

[24] DENNING, D. E. An intrusion-detection model. *IEEE Transactions on software engineering*, 2 (1987), 222–232.

[25] DEPREN, O., TOPALLAR, M., ANARIM, E., AND CILIZ, M. K. An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks. *Expert systems with Applications 29*, 4 (2005), 713–722.

[26] EGELE, M., SCHOLTE, T., KIRDA, E., AND KRUEGEL, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR) 44*, 2 (Mar. 2012), 6:1–6:42.

[27] F-SECURE. F-Secure Rapid Detection & Response. [Online], Available: `https://www.f-secure.com/en/web/business_global/rapid-detection-and-response`, Accessed on: March 3, 2019.

[28] F-SECURE. Detecting Targeted Attacks with Broad Context Detection. Tech. rep., F-Secure, 2018. [Online], Available: `https://www.f-secure.com/documents/10192/2317861/F-Secure_Broad_Context_Detection_whitepaper-web.pdf`, Accessed on: February 25, 2019.

[29] F-SECURE. F-Secure Rapid Detection & Response Service. Tech. rep., F-Secure, 2018. [Online], Available: `https://www.f-secure.com/documents/10192/2310496/RDS-Service-Overview.pdf`, Accessed on: February 15, 2019.

[30] FOLEY, M. J. Microsoft drops Dryad; puts its big-data bets on Hadoop. All About Microsoft, ZDNet, Nov

2011. [Online], Available: `https://www.zdnet.com/article/microsoft-drops-dryad-puts-its-big-data-bets-on-hadoop/`, Accessed on: March 25, 2019.

[31] Gandotra, E., Bansal, D., and Sofat, S. Malware analysis and classification: A survey. *Journal of Information Security 5*, 02 (2014), 56.

[32] GitHub. dask / dask. [Online], Available: `https://github.com/dask/dask`, Accessed on: March 21, 2019.

[33] GitHub. MicrosoftResearch / Dryad. [Online], Available: `https://github.com/MicrosoftResearch/Dryad`, Accessed on: March 25, 2019.

[34] GitHub. mrry / ciel. [Online], Available: `https://github.com/mrry/ciel`, Accessed on: March 25, 2019.

[35] GitHub. ray-project / ray. [Online], Available: `https://github.com/ray-project/ray`, Accessed on: March 26, 2019.

[36] Griffin, K., Schneider, S., Hu, X., and Chiueh, T. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2009), Springer Berlin Heidelberg, pp. 101–120.

[37] Ian Ward. Project Jupyter. [Online], Available: `https://jupyter.org/`, Accessed on: April 17, 2019.

[38] IBM and Ponemon Institute. Cost of a data breach study. Tech. rep., 2018. [Online], Available: `https://www.ibm.com/security/data-breach`, Accessed on: June 17, 2019.

[39] Idika, N., and Mathur, A. P. A survey of malware detection techniques. *Purdue University 48* (2007).

[40] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review* (2007), vol. 41, ACM, pp. 59–72.

[41] Kargupta, H., Bhargava, R., Liu, K., Powers, M., Blair, P., Bushra, S., Dull, J., Sarkar, K., Klein, M., Vasa, M., and Handy, D. VEDAS: A mobile and distributed data stream mining system for real-time vehicle monitoring. In *Proceedings of the 2004 SIAM International Conference on Data Mining* (Apr 2004), pp. 300–311.

[42] Kargupta, H., Park, B.-H., Pittie, S., Liu, L., Kushraj, D., and Sarkar, K. Mobimine: Monitoring the stock market from a PDA. *ACM SIGKDD Explorations Newsletter 3*, 2 (2002), 37–46.

[43] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., et al. Jupyter Notebooks — a publishing format for reproducible computational workflows. In *ELPUB* (2016), pp. 87–90.

[44] Konečný, J., Brendan McMahan, H., Ramage, D., and Richtárik, P. Federated optimization: Distributed machine learning for on-device intelligence. *CoRR* (Oct 2016).

[45] Konečný, J., McMahan, B., and Ramage, D. Federated optimization: Distributed optimization beyond the datacenter. *CoRR* (Nov 2015).

[46] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI* (2014), vol. 14, pp. 583–598.

[47] Li, M., Andersen, D. G., Smola, A., and Yu, K. Communication efficient distributed machine learning with the parameter server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1* (Cambridge, MA, USA, 2014), NIPS'14, MIT Press, pp. 19–27.

[48] Mckinney, W. Data structures for statistical computing in Python. *Proceedings of the 9th Python in Science Conference* (01 2010).

[49] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. MLlib: Machine learning in Apache Spark. *The Journal of Machine Learning Research 17*, 1 (2016), 1235–1241.

[50] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 561–577.

[51] Moser, A., Kruegel, C., and Kirda, E. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)* (Dec 2007), pp. 421–430.

[52] Mukherjee, B., Heberlein, L. T., and Levitt, K. N. Network intrusion detection. *IEEE network 8*, 3 (1994), 26–41.

[53] Mukkamala, S., Sung, A. H., and Abraham, A. Intrusion detection using an ensemble of intelligent paradigms. *Journal of Network and Computer Applications 28*, 2 (2005), 167–182. Computational Intelligence on the Internet.

[54] Murray, D. G., and Hand, S. Scripting the cloud with Skywriting. *HotCloud 10* (2010).

[55] Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation* (2011), pp. 113–126.

[56] Nedic, A., and Ozdaglar, A. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control 54*, 1 (2009), 48–61.

[57] Nedic, A., Ozdaglar, A., and Parrilo, P. A. Constrained consensus and optimization in multi-agent networks. *IEEE Transactions on Automatic Control 55*, 4 (2010), 922–938.

[58] Oliphant, T. E. *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[59] Park, B.-h., and Kargupta, H. Distributed data mining: Algorithms, systems, and applications. *Data Mining Handbook* (2002), 341–358.

[60] Peddabachigari, S., Abraham, A., Grosan, C., and Thomas, J. Modeling intrusion detection system using hybrid intelligent systems. *Journal of network and computer applications 30*, 1 (2007), 114–132.

[61] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. scikit-learn: Machine learning in Python. *Journal of machine learning research 12*, Oct (2011), 2825–2830.

[62] Pérez, F., and Granger, B. E. IPython: a system for interactive scientific computing. *Computing in Science & Engineering 9*, 3 (2007), 21–29.

[63] Peteiro-Barral, D., and Guijarro-Berdiñas, B. A survey of methods for distributed machine learning. *Progress in Artificial Intelligence 2*, 1 (Mar 2013), 1–11.

[64] Provost, F., and Hennessy, D. Distributed machine learning: scaling up with coarse-grained parallelism. *Proceedings of 2nd International Conference on Intelligent Systems for Molecular Biology 2* (Feb 1994), 340–7.

[65] Provost, F., and Kolluri, V. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery 3*, 2 (June 1999), 131–169.

[66] Python Packaging Authority and Python Software Foundation. pip – The Python Package Installer. [Online], Available: `https://pip.pypa.io/en/stable/`, Accessed on: April 18, 2019.

[67] Rieck, K., Trinius, P., Willems, C., and Holz, T. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security 19*, 4 (Dec. 2011), 639–668.

[68] Rocklin, M. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference* (2015), pp. 126–132.

[69] Sathyanarayan, V. S., Kohli, P., and Bruhadeshwar, B. Signature generation and detection of malware families. In *Australasian Conference on Information Security and Privacy* (2008), Springer, pp. 336–349.

[70] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems 38*, 1 (2012), 161–190.

[71] Shashanka, M., Shen, M.-Y., and Wang, J. User and entity behavior analytics for enterprise security. In *2016 IEEE International Conference on Big Data (Big Data)* (2016), IEEE, pp. 1867–1874.

[72] Snapp, S. R., Brentano, J., Dias, G. V., Goan, T. L., Heberlein, L. T., Ho, C.-L., Levitt, K. N., Mukherjee, B., Smaha, S. E., Grance, T., Teal, D. M., and Mansur, D. DIDS (Distributed Intrusion Detection System) – Motivation, Architecture, and An Early Prototype. In *Proceedings of the 14th national computer security conference* (1991), vol. 1, Washington, DC, pp. 167–176.

[73] Sommer, R., and Paxson, V. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy* (May 2010), pp. 305–316.

[74] Tan, M. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *In Proceedings of the Tenth International Conference on Machine Learning* (1993), Morgan Kaufmann, pp. 330–337.

[75] Tesauro, G. J., Kephart, J. O., and Sorkin, G. B. Neural networks for computer virus recognition. *IEEE Expert 11*, 4 (Aug 1996), 5–6.

[76] The Apache Software Foundation. Apache Hadoop. `https://hadoop.apache.org`.

[77] The Apache Software Foundation. Apache Mahout. [Online], Available: `https://mahout.apache.org/`.

[78] The Apache Software Foundation. The Apache Software Foundation Announces Apache™ Spark™ as a Top-Level Project. The Apache Software Foundation Blog, Feb 2014. [Online], Available: `https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50`, Accessed on: March 21, 2019.

[79] The Ray Team. Cluster Setup and Auto-Scaling. [Online], Available: `https://ray.readthedocs.io/en/latest/autoscaling.html`, Revision: c6f12e52, Accessed on: March 27, 2019.

[80] The Ray Team. Serialization in the Object Store. [Online], Available: `https://ray.readthedocs.io/en/latest/serialization.html`, Revision: c6f12e52, Accessed on: March 27, 2019.

[81] Tsai, C.-F., Hsu, Y.-F., Lin, C.-Y., and Lin, W.-Y. Intrusion detection by machine learning: A review. *Expert Systems with Applications 36*, 10 (2009), 11994–12000.

[82] Tyugu, E. Artificial intelligence in cyber defense. In *2011 3rd International Conference on Cyber Conflict* (Jun 2011), pp. 1–11.

[83] Vanhaesebrouck, P., Bellet, A., and Tommasi, M. Decentralized collaborative learning of personalized models over networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)* (2017).

[84] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. Graphx: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 2.

[85] Ye, Y., Li, T., Adjeroh, D., and Iyengar, S. S. A survey on malware detection using data mining techniques. *ACM Computing Surveys 50*, 3 (2017), 41.

[86] You, I., and Yim, K. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications* (2010), IEEE, pp. 297–300.

[87] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Kunda, P. K., and Currey, J. DryadLINQ: A system for general-

purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR 8* (2009).

[88] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. *HotCloud 10*, 10-10 (2010), 95.

[89] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles* (2013), ACM, pp. 423–438.

[90] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. Apache Spark: a unified engine for big data processing. *Communications of the ACM 59*, 11 (2016), 56–65.